DW_lp_fifoctl_1c_df

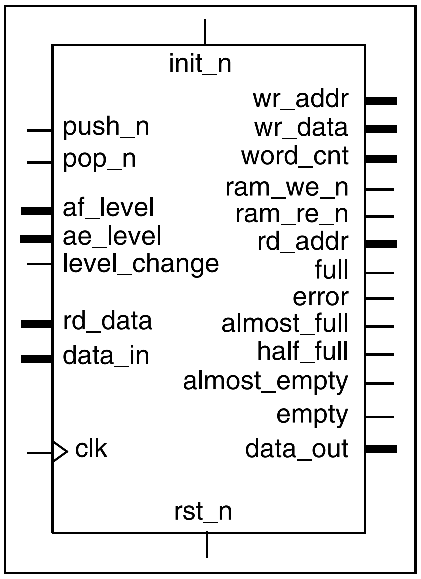
Low Power Single-clock FIFO Controller with Dynamic Flags

Version, STAR, and myDesignWare Subscriptions: IP Directory

Low Power FIFO Controller Family Features and Benefits

Revision History

- RAM bypassing using pre-fetch cache structure
- Single clock cycle execution on all operations
- Low power features and design techniques applied
- Fully registered synchronous address and flag output ports
- Dynamically programmable almost_full and almost_empty flags
- Interfaces to common hard macro or compiled ASIC dual-port synchronous RAMs
- Parameterized RAM size
- Parameterized full-related and empty-related flag thresholds per clock domain
- Push error (overflow) and pop error (underflow) flags per clock domain



Description

This FIFO controller solution is fully configurable to accommodate a wide variety of synchronous RAM architectures including flexibility that is built in to alleviate timing issues in both the `push_n` and `pop_n` interfaces.

Also, this component offers many low-power features that are built in. For example, providing RAM read enabling control, applying RTL implementation techniques conducive to minimizing power, and providing alternative caching architectures.

DW_lp_fifoctl_1c_df is a single-clock FIFO controller intended to interface with dual-port synchronous RAMs. Word caching (or pre-fetching) is performed to minimize latencies via a RAM by-pass feature which allows for bursting of contiguous words and providing registered data to the external logic. The caching *depth* is configurable from 1 to 3 depending on the synchronous RAM configuration. This component contains complete flexibility in interfacing with multiple configurations of synchronous RAM as described by the following.

Supported synchronous RAM architectures with a synchronous write port and any of the following:

- Non re-timed write port and asynchronous read port
- Re-timed write port and asynchronous read port
- Non re-timed write port and synchronous read port with buffered read address and non-buffered read data

- Non re-timed write port and synchronous read port with non-buffered read address and buffered read data
- Non re-timed write port and synchronous read port with buffered read address and buffered read data
- Re-timed write port and synchronous read port with buffered read address and non-buffered read data
- Re-timed write port and synchronous read port with non-buffered read address and buffered read data
- Re-timed write port and synchronous read port with buffered read address and buffered read data

The FIFO controller generates RAM write addressing and write enable logic (push interface), RAM read enable and read addressing (`pop_n` interface), and a comprehensive set of status flags (`empty`, `almost_empty`, `half_full`, `almost_full`, and `full`) and operation error detection logic.

The FIFO controller provides parameterized data *width*, FIFO *depth* (up to 28 address bits or 268435456 locations), pop data pre-fetching cache depth, `almost_empty` and `almost_full` levels that are all configurable upon module instantiation.

As an extra level of flexibility, the DW_lp_fifoctl_1c_df is configurable to allow write path re-timing (push interface) and no pre-fetching cache to model predecessor DesignWare library components.

Attention Unless otherwise stated, the term FIFO means the grouping of the RAM module and pre-fetching cache.

Table 1-1 Pin Description

Pin Name	Width (bits)	Direction	Function
clk	1 bit	Input	Input clock
rst_n	1 bit	Input	Asynchronous reset (active low)
init_n	1 bit	Input	Synchronous reset (active low)
ae_level	ceil($\log_2$[*ram_depth* + 1])	Input	Almost empty level for the `almost_empty` output (the number of words in the FIFO at or below which the `almost_empty` flag is active)
af_level	ceil($\log_2$[*ram_depth* + 1])	Input	Almost full level for the `almost_full` output (the number of empty memory locations in the FIFO at which the `almost_full_s` flag is active).
level_change	1 bit	Input	Enable update of `almost_empty` and/or `almost_full` state when `ae_level` and/or `af_level` change
push_n	1 bit	Input	Push request (active low)
data_in	width bits	Input	Input data
pop_n	1 bit	Input	Pop request (active low)

Table 1-1 Pin Description (Continued)

Pin Name	Width (bits)	Direction	Function
rd_data	width bits	Input	Data read from RAM
ram_we_n	1 bit	Output	Write enable to RAM (active low)
wr_addr	ceil(log2(ram_size*))	Output	Write address to RAM (registered)
wr_data	width	Output	Data written to RAM
ram_re_n	1 bit	Output	Read enable to RAM (active low)
rd_addr	ceil(log2(ram_size*))	Output	Read address to RAM (registered)
data_out	width	Output	Output data
word_cnt	ceil(log2(depth + 1))	Output	FIFO word count
empty	1 bit	Output	FIFO empty flag
almost_empty	1 bit	Output	Almost empty flag (determined by `ae_level` input)
half_full	1 bit	Output	Half full flag
almost_full	1 bit	Output	Almost full flag (determined by `af_level` input)
full	1 bit	Output	Full flag
error	1 bit	Output	Error flag (overrun or underrun)

Attention

* "ram_size" is not a user parameter but is used here as a reference which is derived from the parameters *depth*, *mem_mode*, and *arch_type* as defined in the following:

- If *arch_type* = 0, then "ram_size" = *depth*
- If *arch_type* = 1 or 3 (pre-fetch cache and no input re-timing), see Table 1-2
- If *arch_type* = 2 or 4 (pre-fetch cache and input re-timing), see Table 1-3 on page 4

Table 1-2 Calculation of RAM Size for *arch_type* = 1 or 3

ram_size value based on depth and mem_mode
ram_size = *depth*-1 when *mem_mode* = 0[a]
ram_size = *depth*-2 when *mem_mode* = 1, 2, 4 or 6
ram_size = *depth*-3 when *mem_mode* = 3, 5 or 7

a. Applicable only for *arch_type* of 1. An *arch_type* of 3 is not valid when *mem_mode* is 0.

If *arch_type* is 2 or 4 (pre-fetch cache and input re-timing), then:

Table 1-3 Calculation of RAM Size for *arch_type* of 2 or 4

ram_size value based on depth and mem_mode
ram_size = *depth*-2 when *mem_mode* = 0[a]
ram_size = *depth*-3 when *mem_mode* = 1, 2, 4 or 6
ram_size = *depth*-4 when *mem_mode* = 3, 5 or 7

a. Only applicable for *arch_type* of 2. An *arch_type* of 4 is not valid when *mem_mode* is 0.

Table 1-4 Parameter Descriptions

Parameter	Values	Function
width	1 to 4096 Default: 8	Vector width of “data” bus to/from RAM
depth	4 to 2^{28} Default: 8	Depth of the FIFO.
mem_mode[a]	0 or 7 Default: 3	RAM configuration Identifies where and how many re-timing stages needed in RAM which determines pre-fetch cache buffering depth ■ 0: No pre or post retiming ■ 1: RAM data out re-timing ■ 2: RAM read address re-timing ■ 3: RAM data out and read address re-timing ■ 4: RAM write interface re-timing ■ 5: RAM write interface and RAM data out re-timing ■ 6: RAM write interface and read address re-timing ■ 7: RAM write interface, read address, and read address re-timing For details about *mem_mode*, see “Detailed Description of mem_mode Setting” on page 6

Table 1-4 Parameter Descriptions (Continued)

Parameter	Values	Function
arch_type[a]	0 to 4 Default: 1	Datapath architecture configuration ■ 0: No input re-timing, no pre-fetch cache (*mem_mode* must be set to 0 for this setting of *arch_type*) ■ 1: No input re-timing, pipeline pre-fetch cache ■ 2: Input re-timing, pipeline pre-fetch cache ■ 3: No input re-timing, register file pre-fetch cache (only valid when *mem_mode* $\geq$ 1) ■ 4: Input re-timing, register file pre-fetch cache (only valid when *mem_mode* $\geq$ 1) For details about *arch_type*, see "Detailed Description of Parameter arch_type" on page 8
af_from_top	0 or 1 Default: 1	Almost full level input (`af_level`) usage ■ 0: The `af_level` input value represents the minimum number of valid FIFO entries at which the `almost_full` output starts being asserted. ■ 1: The `af_level` input value represents the maximum number of unfilled FIFO entries at which the `almost_full` output starts being asserted. For details about *af_from_top*, see "Detailed Description of Parameter af_from_top" on page 8
ram_re_ext	0 or 1 Default: 0	Extend `ram_re_n` during active read through RAM ■ 0: Single-pulse of `ram_re_n` at read event of RAM ■ 1: Extend assertion of `ram_re_n` while active read event traverses through RAM
err_mode	0 or 1 Default: 0	Error reporting ■ 0: Sticky error flag ■ 1: Dynamic error flag

a. For valid *mem_mode* and *arch_type* combinations, see Table 1-7 on page 6.

Table 1-5 Synthesis Implementations

Implementation Name	Function	License Required
rtl	Synthesis model	■ DesignWare (P-2019.03 and later) ■ DesignWare-LP[a] (before P-2019.03)

a. For versions before P-2019.03, you must enable minPower as follows:
```
set_synthetic_library {dw_foundation.sldb dw_minpower.sldb}
```

The following simulation models are available:

Table 1-6 Simulation Models

Model	Function
DW03.DW_LP_FIFOCTL_1C_DF_CFG_SIM	Design unit name for VHDL simulation
dw/dw03/src/DW_lp_fifoctl_1c_df_sim.vhd	VHDL simulation model source code
dw/sim_ver/DW_lp_fifoctl_1c_df.v	Verilog simulation model source code

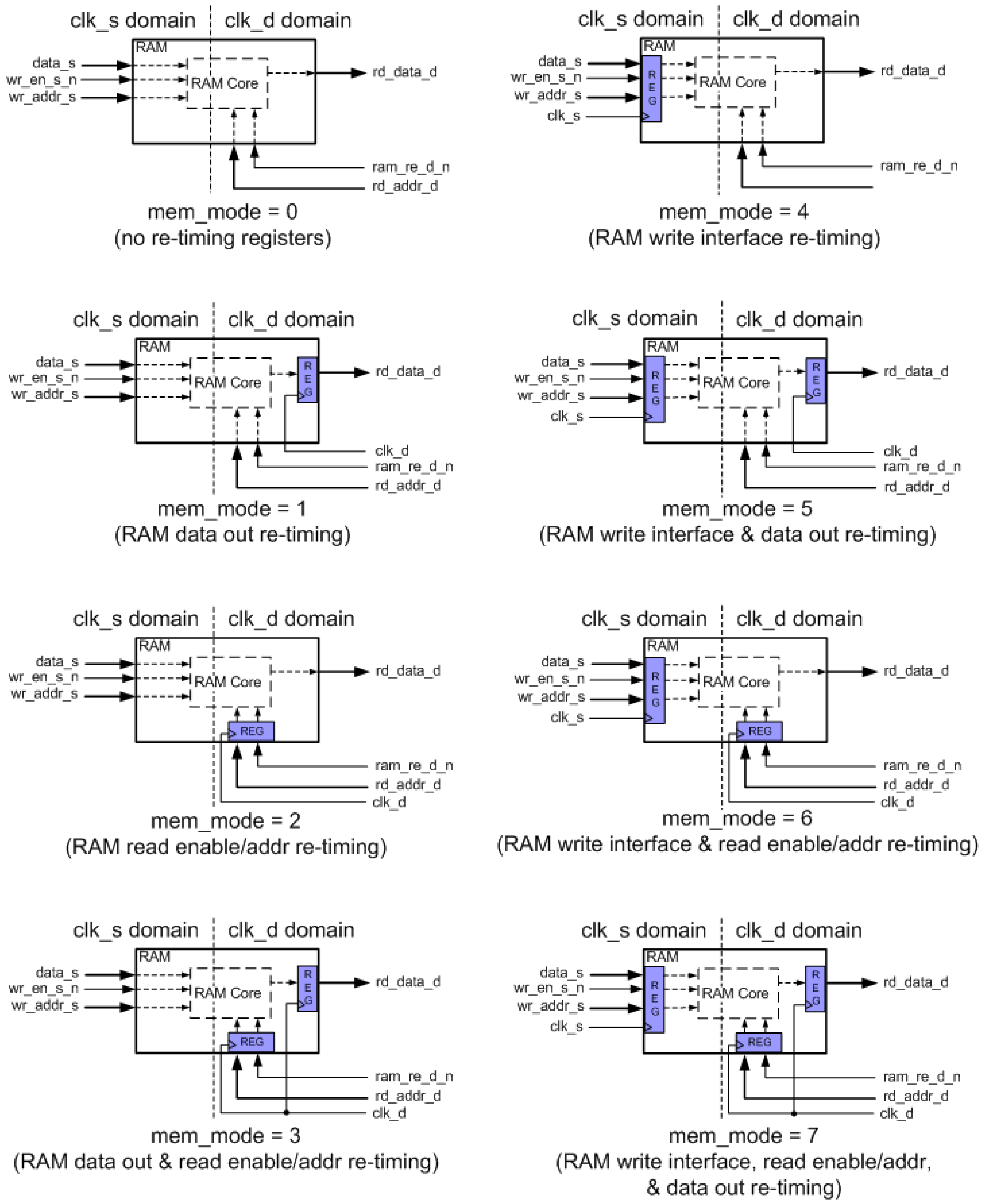
Detailed Description of *mem_mode* Setting

To set the *mem_mode* parameter properly, knowledge of the RAM being used with the DW_lp_fifoctl_1c_df is needed. The following diagrams show the 8 possible RAM architectures that can interface with DW_lp_fifoctl_1c_df and the required *mem_mode* setting for each.

Table 1-7 Legal Settings for the Combination of *mem_mode* and *arch_type* Values

mem_mode values	*arch_type* values	Valid setting?
0	0, 1, and 2	Yes
1 to 7	0	No
0	3 and 4	No
1 to 7	1 to 4	Yes

Figure 1-1 ***mem_mode*** **Settings based on RAM Architecture**



Detailed Description of Parameter *arch_type*

The *arch_type* parameter is available for selection of which structures will exist, if any, in the data path before and after the RAM. If `arch_type` is 0, the DW_lp_fifoctl_1c_df will route the data path and push and pop controls directly to and from the RAM. That is, no storage elements will exist in the data path of the DW_lp_fifoctl_1c_df.

When *arch_type* is non-zero, the DW_lp_fifoctl_1c_df contains a pre-fetch cache used as a RAM bypass to minimize latencies. The depth of the pre-fetch cache is dependent on the synchronous RAM configuration (see Table 1-10 on page 13).

When *arch_type* is 2 or 4, a one stage set of input re-timing registers is placed on `data_in` and `push_n`. This input re-timing stage enables flexibility for designs with a late-arriving write interface and provides the designer a built-in structure to meet timing constraints.

There are two pre-fetch architectures available to allow the designer to optimize for power consumption considerations; *arch_type* settings 1 and 2 vs. 3 and 4. If reduction of power consumption is a system guideline, selecting the optimal pre-fetch cache architecture is dependent on the push and pop activity characteristics to the FIFO. More detail is provided in section "Pre-fetch Cache architectures" on page 9. For legal *arch_type* parameter values in combination with *mem_mode* values see Table 1-7 on page 6.

Detailed Description of Parameter *af_from_top*

The *af_from_top* provides the option for how the `af_level` input is interpreted and used in determining how the state of the `almost_full` output is derived. When *af_from_top* is 1 (default) the DW_lp_fifoctl_1c_df interprets the `af_level` value as the maximum number of empty locations in the FIFO from the top (or full condition) in which the `almost_full` flag is set to 1. The `af_level` value in this case is subtracted from *depth* and compared to `word_cnt`.

If the desire is not to have a subtraction operator built in to the DW_lp_fifoctl_1c_df with regard to determining `almost_full` and, thus, gaining a area savings, setting *af_from_top* to 0 would be the choice to make. For this setting of *af_from_top*, `af_level` is interpreted as the threshold of the word count at which `almost_full` is set to 1. Any word count equal to or greater than `af_level` would result in `almost_full` being a 1, else it would be 0.

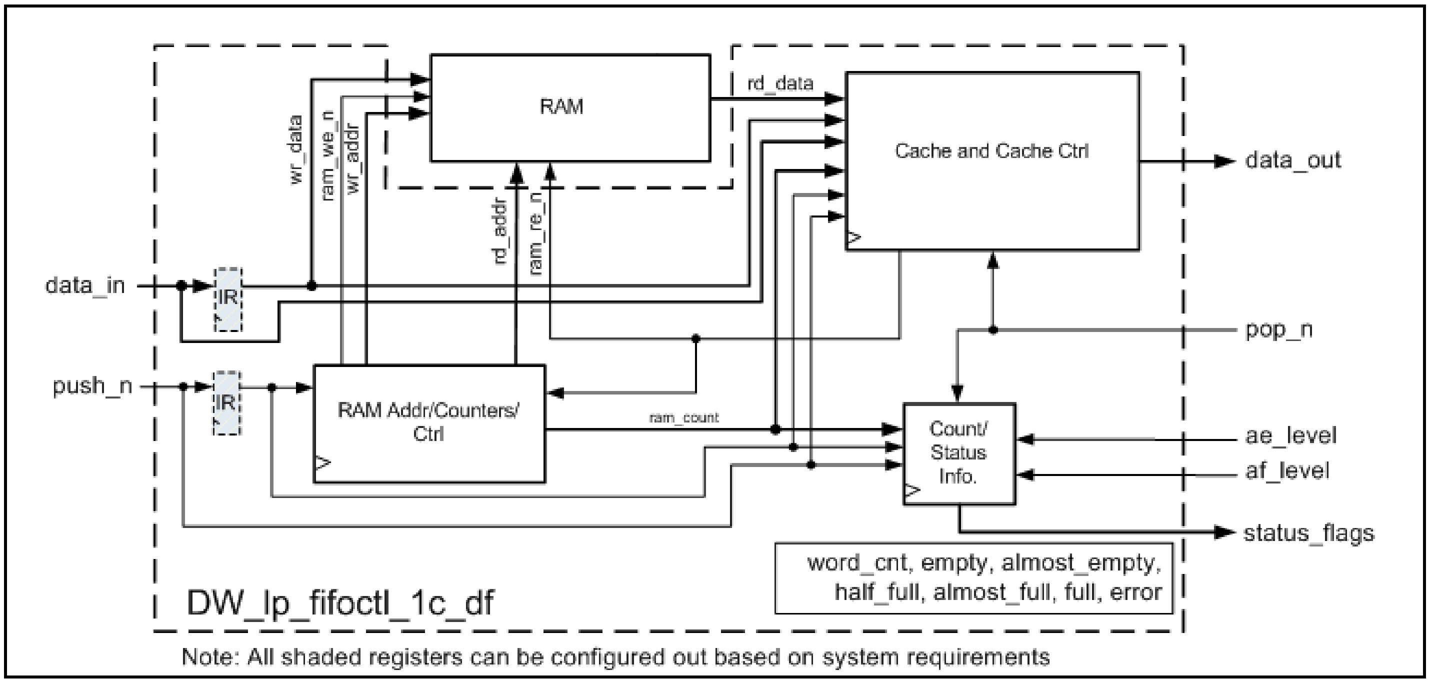
Examples of both settings of *af_from_top* are shown in the "Timing Waveforms" on page 23, for *af_from_top* being 0 refer to Figure 1-7 on page 24, for *af_from_top* being 1, refer to Figure 1-8 on page 25.

Architecture

Block Diagram

Figure 1-2 shows the block diagram for the DW_lp_fifoctl_1c_df component.

Figure 1-2 DW_lp_fifoctl_1c_df Basic Block Diagram



Low Power Features

Emphasis is made on finding solutions that minimize power consumption. Application of component features, RTL implementation techniques, and providing alternative functional architectures are incorporated in this component. The following are items that describe the features implemented in DW_lp_fifoctl_1c_df that focus on power minimization.

RAM Enable

A read enable to RAM is provided to allow the RAM to disable read switching activity when no read access is occurring. A complete FIFO solution in applications where popping is not continuous, shutting down the read port of the RAM during idle popping cycles could assist in minimizing power.

Coding Style enabling Clock Gating

Code implementation techniques are applied throughout that enables the power compiling tool to perform clock gate insertion. This approach provides a significant power savings especially as data widths get larger.

Pre-fetch Cache architectures

There are two pre-fetch cache architectures which are parameter selectable that allows for power optimization: pipelined and register file types.

The 'pipelined' caching style is effectively a shift register of 1, 2, or 3 stages. Active switching through each stage occurs during shifting initiated by pop requests with pending valid data in either the RAM or cache stages behind the head location. In cases with a wide data bus and cache configurations of 2 or 3 deep, this could represent the majority of the register switching power consumption within the component.

As an alternative architecture for cache depths of 2 or 3, the pre-fetch cache is organized as a ‘register file’ structure. For the ‘register file’ cache structure, the shifting between pipelined cache entries is eliminated and replaced with write and read pointer manipulation to access cache elements; a mini-FIFO of sorts.

The two caching architectures are provided to give the designer flexibility in selecting the caching architecture that will yield the lowest total power consumption.

Knowing which pre-fetch cache architecture to choose is highly dependent on factors such as technology, clock rate, data *width*, pre-fetch cache *depth*, and data flow characteristics through the associated FIFO.

With all variables being equal, the advantage that either cache architecture provides in terms of optimal power dissipation is based particularly on the type of data flow through the DW_lp_fifoctl_1c_df.

Generally, there is no rule of thumb in selecting which cache architecture will render the least power dissipation. This may require the design process to include some experimentation in using both to characterize behavior based on the system parameters.

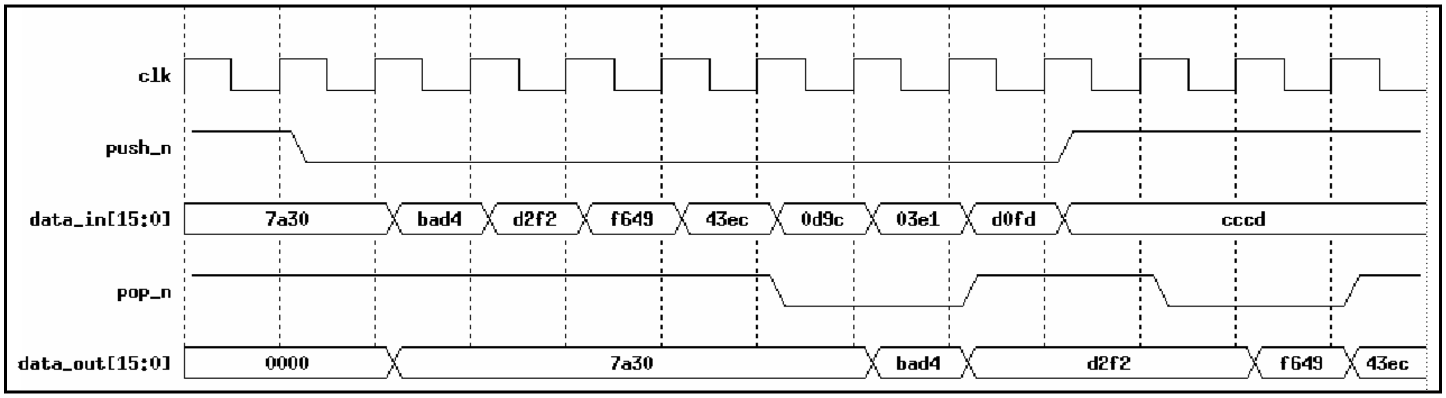
Note Keep in mind, choosing which pre-fetch cache architecture is only meaningful in a system when the parameter *mem_mode* is not 0. That is, when the cache depth is 2 or 3, the selection of the cache architecture would become relevant. Cache depth is determined by the *mem_mode* parameter. Cache depth of 2 or 3 means *mem_mode* is non-zero. When *mem_mode* is 0 and *arch_type* is non-zero, cache depth is 1. When cache depth is 1, only the pipeline structure of pre-fetch cache is available. Therefore, when *mem_mode* is 0, *arch_type* of '3' and '4' are considered illegal and errors in functional simulation and synthesis will be reported. Table 1-7 on page 6 depicts legal *mem_mode* / *arch_type* combinations.

Consider two data flow behaviors (and the cache architecture) that most likely will yield a better power result. Of course, this assumes that this data flow is to be the predominant characteristic through the DW_lp_fifoctl_1c_df.

The key factor that determines which cache architecture will provide the least power consumption depends on the behavior of the pop requests (`pop_n`). If pop requests are issued in short bursts of 3 or less, the Register File cache will most likely yield the least power consumption versus the Pipeline cache architecture. If, however, pop requests occur in longer contiguous bursts, the Pipeline cache architecture is favorable. The following two cases show the two extremes in data flow behavior in which each cache architecture would be selected, generally, in terms of providing optimal power consumption.

CASE 1: Push packets and pop alternately with 2 cycles active then 2 cycles inactive

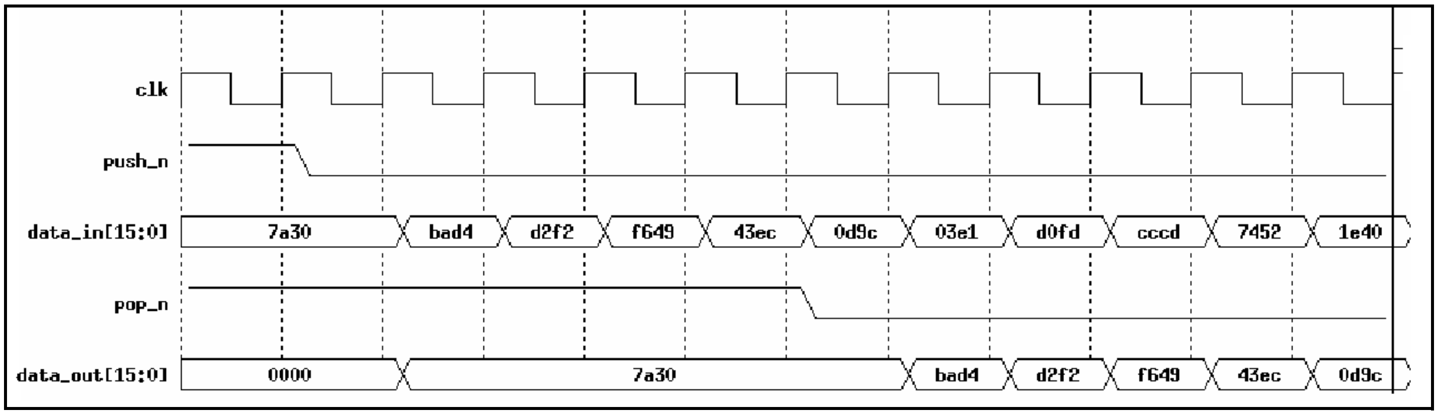
Figure 1-3 Data Flow of Popping Small Bursts



The data flow behavior for Figure 1-3 shows a packet of length *depth* (8 in this case) with words pushed contiguously before pushing halts. The pop activity begins as the FIFO is approximately half full and follows a progression of 2 cycles active and 2 cycles idle. This particular data flow, assuming cache depth is 3, is the best-case behavior that favors the selection of the Register File (RF) cache architecture (*arch_type* of 3 or 4). Similarly, pop request being active every other cycle for cache depth of 2 (*arch_type* of 1 or 3) would be best-case data flow behavior geared to selecting the RF cache style.

CASE 2: Popping in long contiguous bursts

Figure 1-4 Data Flow of Popping Continuously



When long bursts of contiguous pop requests are issued, this type of data flow would be conducive to using the PL cache architecture. The power benefits of using the PL cache over the RF cache in this data flow comes from the fact that the front of the cache is continuously being written to and read from. From the PL cache perspective only one stage of the cache is being used at a time and effectively the other stages of the cache are unused during this time. So, the dynamic power of shifting through many stages of the cache does not occur. In the RF cache, however, data is being written to cache just like in the PL cache case, but the write and read addressing logic is always active. This activity of the write and read addressing logic is the extra power consumption that the RF cache architecture has that the PL cache does not. Thus, the PL cache architecture would be optimal, in general, for this type of data flow.

Note that in CASE 1, the write and read address logic is always active. But the difference is that the data flow through the cache caused by the bursting pop requests is such that the cache is full, almost full, or becoming full. Thus, all the cache locations are shifting in or out data on every cycle. Therefore, power consumption of the cache is predominantly the shifting of data and not the writing and reading address logic. Thus, selecting the RF cache, in general, will provide best power results. The difference in favor of the RF cache over the PL cache in this data flow behavior increases as the data widths increase.

Memory and FIFO Size Considerations

Depending on the constraints of the system design, the RAM size and configuration may be fixed or the FIFO depth may be the fixed. In cases where the FIFO depth is fixed and the RAM size is derived, refer to Table 1-8 and Table 1-9 in determining the proper RAM size to use when the *depth* parameter is pre-determined.

In the cases where RAM size is fixed, for example, to a 2^n size, then the parameter *depth* must be determined based on the RAM configuration. The following tables (based on the *arch_type* setting) indicates the factors involved in calculated *depth* if RAM size is fixed.

If *arch_type* is 0, then *depth* = RAM size.

If *arch_type* is 1 or 3 (no input re-timing, pre-fetch cache exists), then:

Table 1-8 Calculation of *depth* for *arch_type* of 1 or 3

depth* value based on RAM Size and *mem_mode
depth = RAM size +1 when *mem_mode* = 0[a]
depth = RAM Size + 2 when *mem_mode* = 1, 2, 4 or 6
depth = RAM Size + 3 when *mem_mode* = 3, 5 or 7

a. Only applicable for *arch_type* of 1. An *arch_type* of 3 is not valid when *mem_mode* is 0.

If *arch_type* is 2 or 4 (input re-timing and pre-fetch cache exist), then:

Table 1-9 Calculation of *depth* for *arch_type* of 2 or 4

depth* value based on RAM Size and *mem_mode
depth = RAM size +2 when *mem_mode* = 0[a]
depth = RAM Size + 3 when *mem_mode* = 1, 2, 4 or 6
depth = RAM Size + 4 when *mem_mode* = 3, 5 or 7

a. Only applicable for *arch_type* of 2. *arch_type* of 4 is not valid when *mem_mode* is 0.

Write Operations

Caching (When *arch_type* Is Not 0)

The pop interface contains output buffering (pre-fetching cache) with the number of pipeline stages determined by the *mem_mode* parameter.

When the FIFO is empty, the first word that is pushed goes directly to the pre-fetching cache (bypassing RAM) and is available for reading (popping) in the next clock cycle. All subsequent pushes go directly into the cache until it becomes full. Once the cache is full and another push occurs without a simultaneous pop request, the first RAM location is written. All subsequent pushes will go into the RAM regardless of the cache state until the RAM becomes completely empty. At that time, pushes can begin to fill the cache first prior to any writes into RAM as before. Also, note that a simultaneous push and pop with the cache full and the RAM empty causes push data to bypass the RAM and get loaded into the cache.

The pre-fetching cache can be omitted in certain configurations as needed by the system designer. However, in omitting the pre-fetching cache, there is no guarantee of a registered output from the FIFO.

When the pre-fetching cache is configured to exist, at a minimum, there will always be one buffering stage in the cache which is seen at the pop interface.

Table 1-10 is a list identifying the number of pre-fetching stages of the cache used based on the value of the *mem_mode* parameter.

Table 1-10 Cache Size When *arch_type* is 1 Through 4

mem_mode Values	Number of Caching Stages
0[a]	1
1, 2, 4, or 6	2
3, 5, or 7	3

a. Exception: *arch_type* values of 3 and 4 are not valid when *mem_mode* is 0.

Writing to RAM

The `wr_addr` and `ram_we_n` output ports of the FIFO controller provide the write address and synchronous write enable, respectively, to the RAM.

When the condition is met for writing to RAM as described in "Caching (When arch_type Is Not 0)", a write to RAM is executed when the `push_n` input is asserted and either:

- the `full` flag is inactive, or:
- the `full` flag is active *and* the `pop_n` is asserted.

Thus, a push can occur even if the FIFO is full as long as the pop is executed in the same cycle.

Asserting `push_n` when `full` is not asserted causes the following to occur when *arch_type* is 0, 1, or 3, the `ram_we_n` is asserted immediately, preparing for a write to the RAM on the next rising `clk`, and on the next rising edge of `clk`, `wr_addr` is advanced.

When *arch_type* is 2 or 4, `ram_we_n` and `wr_addr` would be delayed by one clock cycle due the existence of input re-timing registers on the `push_n` and `data_in`.

Thus, the RAM is written and `wr_addr` (which always points to the address of the next word to be pushed) is incremented on the same rising edge of `clk` - the first clock after `ram_we_n` is asserted. This means that `push_n` must be asserted early enough to propagate through the FIFO controller to the RAM before the ensuing clock cycle.

In systems where the `push_n` and/or associated `data_in` are late-arriving, the DW_lp_fifoctl_1c_df is configurable to add one level of re-timing registers to the both sets of signals to guarantee meeting timing specifications as configured by the *arch_type* parameter.

Write Errors

An error occurs if a push operation is attempted while the FIFO is `full`. That is, the `error` output goes active if:

- the `push_n` input is asserted,
- the `pop_n` input is not asserted, and
- the `full` flag is active on the rising edge of `clk`.

When a push error occurs, `ram_we_n` stays inactive (high) and the write address, `wr_addr`, does not advance. After a push error, although a data word was lost at the time of the error, the FIFO remains in a valid `full` state and can continue to operate properly with respect to the data that was contained in the FIFO before the push error occurred.

Read Operations

Reading from RAM

The read port of the RAM can be asynchronous or synchronous with clock. All read data from RAM is first loaded into the pre-fetching cache when *arch_type* is not 0. The `rd_addr` output port of the DW_lp_fifoctl_1c_df provides the read address to the RAM. `rd_addr` points to (pre-fetches) the next word of RAM read data to be loaded to cache. Reading of RAM is initiated when `pop_n` is asserted while the cache contains at least one valid data entry and the RAM is not empty.

Asserting `pop_n` while `empty` is not active causes the internal read pointer to increment on the next rising edge of `clk` only if the RAM contains at least one valid entry. Therefore, for asynchronous read port memories, the RAM read data must be captured in the cache on the rising edge of `clk` following the assertion of `pop_n`.

For synchronous read port memories, when either `rd_addr` or RAM data out (`rd_data`) is buffered, data is captured by the cache on the rising edge of `clk` one cycle after the `clk` edge that directed the controller to read, or when both `rd_addr` and RAM data out are buffered then data is captured by the cache on the rising edge of `clk` two cycles after the initiating `clk` edge that directed the controller to read.

If the RAM is empty at the assertion of `pop_n`, the internal read pointer does not advance.

Popping from the Cache (Referencing Pipelined Architecture)

The cache is the data interface of the FIFO (when *arch_type* is not 0) and it is made up of pipelined data words based on the *mem_mode* parameter as described in Table 1-10 on page 13. When the head of the cache (the `data_out` output port) contains a valid entry the FIFO is considered not empty (the `empty` flag is not asserted), a legal pop of the FIFO is allowed (asserting `pop_n`). When `empty` is not asserted the `data_out` contents is the next valid word from the FIFO. The assertion of `pop_n` causes the cache pipeline to shift valid data, if any, on the next rising edge of `clk`. If active RAM data out (`rd_data`) is available, `rd_data` is loaded into the closest vacated stage to the head of the cache. For example, if only the head stage of the cache contains valid data and `rd_data` is valid and `pop_n` is asserted, then on the next rising edge of `clk` the `rd_data` is loaded to the head of the cache. This event would keep `empty` de-asserted and allow for another pop on the next rising edge of `clk`.

However, if only the head of the cache contains valid data and `rd_data` is not valid, `push_n` is not asserted, and `pop_n` is asserted, then on the next rising edge of `clk`, the data value at the head of the cache (`data_out`) is held, but the `empty` flag gets asserted and `word_cnt` goes to 0. Thus, assertion of the `empty` flag declares the contents at `data_out` irrelevant.

Read Errors

An error occurs if a pop operation is attempted while the FIFO is empty. That is, the `error` output goes active if:

- the `pop_n` input is active and
- the `empty` flag is active on the rising edge of `clk`.

When a pop error occurs, the read address, `rd_addr`, does not advance. After a pop error the FIFO is still in a valid empty state and can continue to operate properly.

Status Flags and Error Output

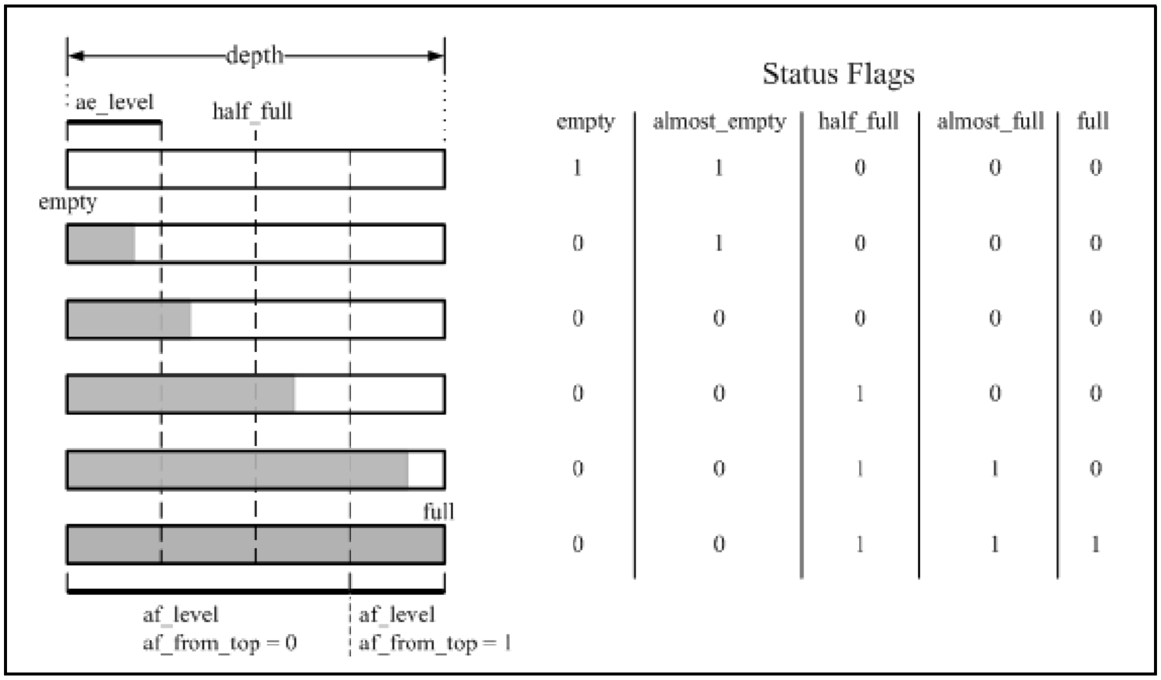
The error outputs and flags are initialized as follows:

- `empty` and `almost_empty` are initialized to 1
- all other flags and the `error` output are initialized to 0

The `almost_empty` flag is dependent on the input value of `ae_level`. Similarly, the `almost_full` flag is dependent on the value placed on the input `af_level` as depicted in Figure 1-5 on page 16. When `ae_level` and `af_level` are held fixed during normal operation, the `almost_empty`, `almost_full`, and the other status flags respond immediately to changes in state on the next rising edge of clock following either a push or pop operation. If, however, the FIFO is in a non-empty state and either or both of the inputs `ae_level` and `af_level` change, the corresponding state of `almost_empty` and `almost_full` will not update until either a push or pop request is issued. That is, when `ae_level` and/or `af_level` input are changed, there could be a moment in time when the `almost_empty` and/or `almost_full` flags are not accurate when compared to the word count reported. The `almost_empty` and/or `almost_full` flags will be correct again on the next rising edge of `clk` after either `push_n` or `pop_n` (but not both) is asserted. If this behavior is not desired, the `level_change` input is provided to enable the updating of the `almost_empty` and/or `almost_full` flags on the next rising edge of `clk` following the change of the `ae_level` and/or `af_level` inputs while `level_change` is active (1).

This method of updating of status flags only when the word count changes (push without pop or pop without push) is in place to minimize dynamic power in the DW_lp_fifoctl_1c_df. So, providing the `level_change` input enables newly changed `ae_level` and `af_level` input values to be immediately applied to report accurate status flags without relying on active push or pop requests. However, if this behavior is not critical, tying off the `level_change` input to a logic 0 will provide better quality of results in timing critical designs. For example, if system behavior exists where `ae_level` and `af_level` values are only changed during system reset, then it is recommended that `level_change` be tied to logic 0 to provide the best possible opportunity to minimize timing through the component.

Figure 1-5 Status Flag Interpretation



empty	almost_empty	half_full	almost_full	full
1	1	0	0	0
0	1	0	0	0
0	0	0	0	0
0	0	1	0	0
0	0	1	1	0
0	0	1	1	1

Most status flags have a property which is potentially useful to the designed operation of the FIFO controller. These properties are described in the following explanations of the flag behaviors.

empty Status Flag

The `empty` output is active high and registered. `empty` indicates that the FIFO contains no valid data entries. During the first push the rising edge of `clk` causes the first word to be written into the pre-fetch cache (or RAM if `arch_type` is 0) and `empty` is driven low. Upon the pop of the last valid data entry (and no simultaneous push request), the `empty` is driven high on the next rising edge of `clk`.

Property of empty

If `empty` is active then the FIFO is truly empty.

almost_empty Status Flag

The `almost_empty` output is active high, registered, and indicates that the FIFO is almost empty when there are no more than `ae_level` (input port) words currently in the FIFO to be popped.

The `ae_level` input defines the almost empty threshold. The `almost_empty` output is useful when it is desirable to push data into the FIFO in bursts (without allowing the FIFO to become empty).

Property of almost_empty

If `almost_empty` is active then the FIFO has at least '*depth* - `ae_level`' available locations. Therefore such status indicates that the push interface can safely and unconditionally push (*depth* - `ae_level)` words into the FIFO. This property guarantees that such a 'blind push' operation will not overrun the FIFO.

half_full Status Flag

The `half_full` output is active high, registered, and indicates that the FIFO has at least half of its memory locations occupied.

Property of half_full

If `half_full` is inactive then the FIFO has at least half of its locations available. Thus such status indicates that the push interface can safely and unconditionally push 'INT(*depth*/2)' words into the FIFO. This property guarantees that such a 'blind push' operation will not overrun the FIFO.

almost_full Status Flag

The `almost_full` output active high and registered. Depending on the parameter *af_from_top* setting `almost_full` indicates either that the FIFO is almost full when there are no more than `af_level` empty locations in the FIFO (*af_from_top* is 1) or when there is at least `af_level` used locations in the FIFO (*af_from_top* is 0).

The `af_level` input port defines the almost full threshold. The `almost_full` output is useful when more than one cycle of advance warning is needed to stop the flow of data into the FIFO before it becomes full (to avoid a FIFO overrun). Also, it allows for a 'blind pop' operation since it guaranteed to have at least `af_level` entries exist for the *af_from_top* is 0 case or (*depth* - `af_level)` entries exist for the *af_from_top* is 1 case.

When *af_from_top* is set to 1, a subtraction operation is performed using *depth* and `af_level`. When *af_from_top* is setting to 0, no subtraction is performed using the `af_level` input.

Property of almost_full

For *af_from_top* is 0:
If `almost_full` is inactive (low) then the RAM module has at least (*depth* - `af_level`+1) available locations. Thus such status indicates that the push interface can safely and unconditionally push (*depth* - `af_level`+1) words into the FIFO. This property guarantees that such a 'blind push' operation will not overrun the FIFO.

For *af_from_top* is 1:
If `almost_full` is inactive (low) then the RAM module has at least (`af_level`+1) available locations. Thus such status indicates that the push interface can safely and unconditionally push (`af_level`+1) words into the FIFO. This property guarantees that such a 'blind push' operation will not overrun the FIFO.

full Status Flag

The `full` output is active high, registered, and indicates that the FIFO has valid data entries in every location. During the final push the rising edge of `clk` causes the last word to be pushed and `full` is asserted. A pop request when the FIFO is full (and no simultaneous push request) causes the `full` flag to de-assert on the next rising edge of `clk`.

Property of `full`

If the `full` output is active, then all available entries in the FIFO (RAM, cache, and input re-timing stage (if configured to exist)) have valid data in them.

word_cnt

The `word_cnt` output is registered and represents the number of valid data entries in the FIFO. This count includes the contents in the RAM, pre-fetching cache (if configured into the design), and the input re-timing stage (if configured into the design). Upon detection of separate push and pop events, the `word_cnt` gets updated on the next rising edge of `clk`. Simultaneous push and pop events will not change its value.

The range of `word_cnt` is from 0 to *depth*.

error Output

The `error` output can indicate that a push request was issued while the `full` output was active and no pop request (an overrun error) or that a pop request was issued while the `empty` output was active (an underrun error).

The *err_mode* parameter determines whether the `error` output remains active until reset (persistent) or for only the clock cycle(s) in which the error is detected (dynamic).

When the *err_mode* parameter is set to 0 at design time, persistent error flags are generated. When the *err_mode* parameter is set to 1 at design time, dynamic error flags are generated.

When an overrun condition occurs, the write address pointer (`wr_addr`) does not advance, and the RAM write enable (`ram_we_n`) is not activated.

Therefore, a push request that would overrun the FIFO is, in effect, rejected, and an error is generated. This guarantees that no data already in the FIFO is destroyed (overwritten). Other than the loss of the data accompanying the rejected push request, FIFO operation can continue without reset.

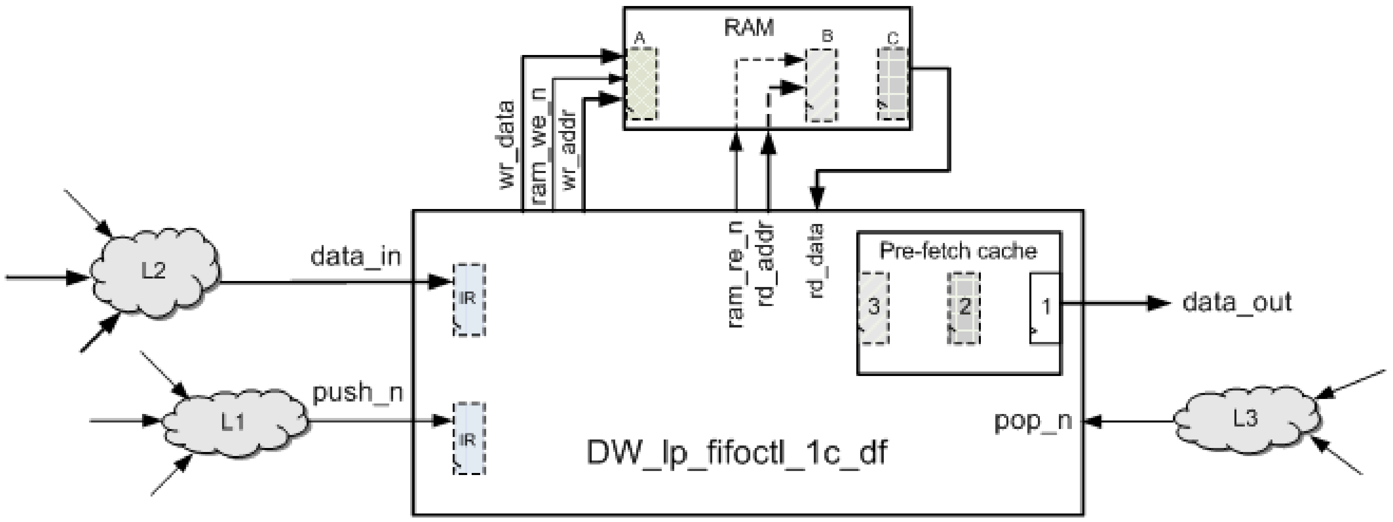
When an underrun condition occurs, the read address pointer (`rd_addr`) does not advance, as there is no data in the FIFO to retrieve.

The FIFO timing is such that the logic controlling the `pop_n` input would not see the error until 'nonexistent' data had already been registered by the receiving logic. This is easily avoided if this logic can pay close attention to the `empty` output and thus avoid an underrun completely.

Considerations for Setting the *mem_mode* Parameter

Depending on the logic being supplied to the RAM that interfaces with the DW_lp_fifoctl_1c_df and the RAM architecture, the setting of the *mem_mode* parameter is determined. Refer to Figure 1-6 for the following discussion which assumes a pre-fetch pipelined cache.

Figure 1-6 ***mem_mode*** **Settings Based on System Design**



If there contains L1 and/or L2 logic clouds or delays that cause either `push_n` and/or `data_in`, respectively, to be late-arriving, then there potentially would be the need to re-time the writing signals into the RAM as denoted by register A. If register A of the RAM is required in the design, the *mem_mode* parameter should be set to 4 or greater depending on the existence of registers B and C in the RAM.

Similarly, if there is an L3 logic cloud or delays that cause `pop_n` to be late-arriving enough to require a register B in the RAM, then *mem_mode* should be set to 2, 3, 6, or 7 depending on the existence of registers A and C in the RAM.

If delay is minimal through L1, L2, and L3 which does not require either register A or B in the RAM, then depending on the internal delay of the datapath through the RAM to its output a re-timing register C may or may not be needed. If needed, then the *mem_mode* must be set to either 1 (register C exists) or 0 (register C does not exist).

Table 1-11 lists the all possible supported RAM architectures and the required *mem_mode* setting along with the resulting structure of the pre-fetch cache.

Table 1-11 RAM Configuration Determines mem_mode Setting

RAM register exists?				
A	**B**	**C**	***mem_mode*** **Setting**	**Pre-fetch Cache Structure**
no	no	no	0	register 1
no	no	yes	1	registers 1 and 2
no	yes	no	2	registers 1 and 2

Table 1-11 RAM Configuration Determines mem_mode Setting (Continued)

RAM register exists?				
A	**B**	**C**	***mem_mode* Setting**	**Pre-fetch Cache Structure**
no	yes	yes	3	registers 1, 2, and 3
yes	no	no	4	registers 1 and 2
yes	no	yes	5	registers 1, 2, and 3
yes	yes	no	6	registers 1 and 2
yes	yes	yes	7	registers 1, 2 and 3

Reset

System Resets (Synchronous and Asynchronous)

Two system resets are available: `rst_n` is asynchronous and `init_n` is synchronous.

If both resets are connected to active logic, `rst_n` has precedence if both resets are asserted simultaneously. If only one of the resets is active in the system, the other should be tied to the de-asserted state (logic high). For examples of timing waveforms where `init_n` and `rst_n` are asserted, see Figure 1-12 on page 30 and Figure 1-13 on page 31.

Status Flags During Reset Conditions

Table 1-12 identifies what state each status flag should be set to when in reset.

Table 1-12 Status Flag States During Reset Conditions

Status Flag	Value During Reset
empty	1
almost_empty	1
half_full	0
almost_full	0
full	0
word_cnt	0
error	0

Simulation Assertions (SystemVerilog only)

The Verilog simulation model and the Cores IP model incorporate SystemVerilog assertions covering functionality. By default, all the assertions have a reporting severity of 'error' but they can be changed to report as 'warning', 'fatal', or 'not at all' by defining the preprocessing variable named DW_SVA_MODE. Not defining DW_SVA_MODE behaves as if it were defined as '2' (report as 'error'). So, for example, if the desire is to have all the assertions in this component report with severity of 'warning' do the following:

```
`define DW_SVA_MODE   1
```

Table 1-13 lists the assertion reporting severity based on the DW_SVA_MODE value. It is important to note the value of DW_SVA_MODE determines the same reporting severity of all the assertions within this component.

Table 1-13 DW_SVA_MODE Settings

DV_SVA_MODE value	Assertion Report Severity
not defined (default)	error
0	disable reporting
1	warning
2	error
3	fatal

The following is a list of key assertions included in this component:

- Word count is within legal range
- Address pointers stay within legal range
- Status flags (full and empty) are in the proper state under system reset conditions
- Full and empty status are in the proper states when not in system reset conditions
- Addresses (and internal pointers) do not change under push and/or pop error cases

Figure 1-2 shows the block diagram of the DW_lp_fifoctl_1c_df. For reference purposes, a RAM is included in the diagram but does not exist in the DW_lp_fifoctl_1c_df component.

Suppressing Warning Messages During Verilog Simulation

The Verilog simulation model includes macros that allow you to suppress warning messages during simulation.

To suppress all warning messages for all DWBB components, define the DW_SUPPRESS_WARN macro in either of the following ways:

- Specify the Verilog preprocessing macro in Verilog code:

```
`define DW_SUPPRESS_WARN
```

- Or, include a command line option to the simulator, such as:

  `+define+DW_SUPPRESS_WARN` (which is used for the Synopsys VCS simulator)

The warning messages for this model include the following:

- If values other than 1 or 0 are present on a clock port, the following message is displayed:

```
WARNING: <instance_path>.<clock_name>_monitor:
    at time = <timestamp>, Detected unknown value, x, on <clock_name> input.
```

  To suppress only this warning message for all DWBB components, use the following macro:

  - Define the DW_DISABLE_CLK_MONITOR macro. You can define this macro in the following ways:
    - Specify the Verilog preprocessing macro in Verilog code:

```
`define DW_DISABLE_CLK_MONITOR
```

    - Or, include a command line option to the simulator, such as:

      `+define+DW_DISABLE_CLK_MONITOR` (which is used for the Synopsys VCS simulator)

  This message is also suppressed using the DW_SUPPRESS_WARN macro explained earlier.

Timing Waveforms

Figure 1-7 shows the configuration where no pre-fetch cache exists (*arch_type* is 0). When *arch_type* is 0, *mem_mode* must also be 0.

Configuration: *width* is 8, *depth* is 8, *mem_mode* is 0, *arch_type* is 0, *af_from_top* is 0, *ram_re_ext* is 1, and *err_mode* is 0; no pre-fetch cache, no re-timing stages in RAM.

When no pre-fetch caching exists, all writing and reading is directly made to and from the synchronous RAM. In this configuration and example, it is assumed that the synchronous RAM does not contain any re-timing stages.

For all valid push requests (ones that don't cause a push `error`), the `push_n` directly drives the `ram_we_n` signal to RAM. Conversely, for all valid pop requests (ones that don't cause a pop `error`), the `pop_n` drives the `ram_re_n`. The `wr_addr` and `rd_addr` signals connected to the RAM represent the current location of RAM that the write and read operations, respectively, will access. Once a write or read operation completes in the current cycle via the assertion of `ram_we_n` or `ram_re_n`, respectively, these registered addresses get incremented to the next RAM location for writing or reading.

If a push `error` occurs, when `push_n` is 0, `full` is 1, and `pop_n` is 1, no write operation is performed to RAM and the word at `data_in` will be lost. This, however, preserves all the previously written data. The same is true if a pop `error` occurs (`pop_n` is 0 and `empty` is 1). When an invalid pop is requested, no read operation is performed. Thus, in both cases of a push or pop `error` the `wr_addr` and `rd_addr`, respectively, will not advance.

This set of waveforms, shows the behavior of the DW_lp_fifoctl_1c_df when pushing the FIFO from empty to full and popping from full back to empty. After popping to the empty state is completed, another single pop request is issued that causes a pop error (`error` going to 1). In this case, `error` stays set to 1 upon the first occurrence of an error since the parameter *err_mode* is set to 0. Also notice that the `rd_addr` did not advance when the pop `error` occurred.

With regard to the other status flags, the `almost_empty` and `almost_full` outputs are configured based on the inputs `ae_level` and `af_level`, respectively. Additionally, for the `almost_full` flag, the parameter *af_from_top* determines how the `af_level` input value is interpreted.

In this example below, *af_from_top* is 0. This means that whatever value is driven on `af_level` will be the threshold at which the `almost_full` flag goes to 1. Any word count less than `af_level` will result in `almost_full` being a 0. In these waveforms `af_level` is '5'. So, whenever `word_cnt` is 5 or greater, `almost_full` is 1.

The `almost_empty` flag always uses the same interpretation of its corresponding `ae_level` input signal value. Whenever the word count in the FIFO is less than or equal to `ae_level`, `almost_empty` is 1. In this example below, `ae_level` is '4'. So, as long as the `word_cnt` is '4' or less, `almost_empty` is 1.

Figure 1-7 No Caching and No RAM Re-timing Configuration

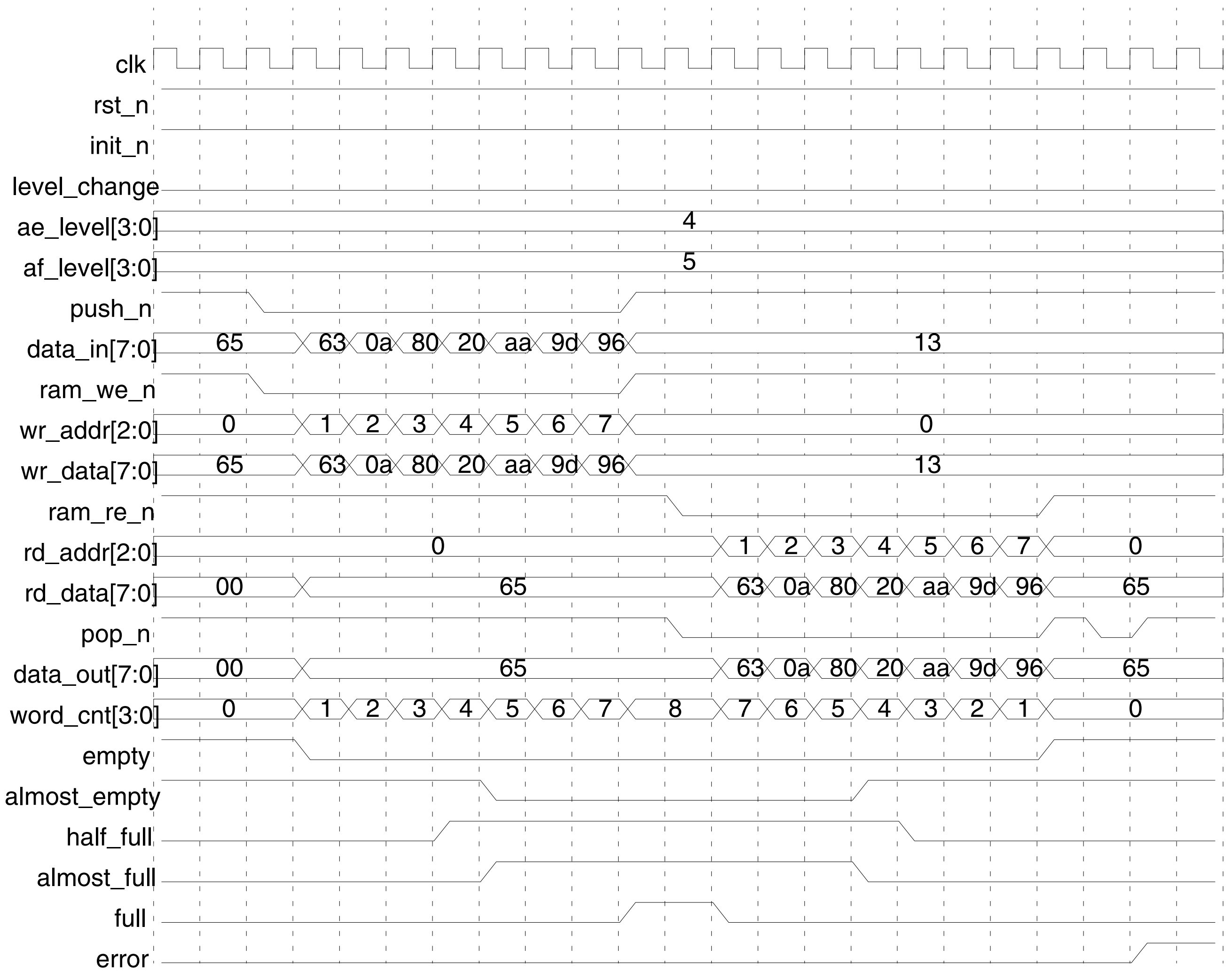


Figure 1-8 illustrates continuously pushing of the FIFO until full from the empty state, multiple cycles of simultaneous push and pop requests during `full` condition, and a push error.

Configuration: *width* is 8, *depth* is 9, *mem_mode* is 3, *arch_type* is 2, *af_from_top* is 1, *ram_re_ext* is 1, and *err_mode* is 1; pre-fetch cache depth of 3, RAM read address and data out re-timing stages.

In this configuration, it is assumed that a synchronous RAM with both read address re-timing and data out re-timing is connected to the DW_lp_fifoctl_1c_df.

When the FIFO is `full` and a push request occurs with no simultaneous pop request, the `error` flag is asserted. The `error` output in this scenario only occurs for single cycle since the push `error` only occurred for one cycle and *err_mode* is 0. If *err_mode* was 1, the `error` output would have remained asserted until a system reset is issued.

With *mem_mode* set to '3' and *arch_type* set to '2', the DW_lp_fifoctl_1c_df contains a 3-deep pre-fetch cache and a single input re-timing register stage for `push_n` and `data_in` before the RAM. Since *mem_mode* is '3', this implies that the RAM must contain both a read address re-timing stage and RAM data output is synchronous. The resulting RAM size is 5 (*depth* - cache depth - one input re-timing stage).

Since *ram_re_ext* is set to 1, the `ram_re_n` signal asserted is extended to track 'active' read operation that makes their way through the re-timing stages of the RAM that eventually become available at `rd_data`.

Note that `almost_empty` de-asserts when `word_cnt` is greater than the value set by `ae_level` which is '2'. `half_full` asserts when `word_cnt` is greater than or equal to '5'. Since *depth* is an odd value, `half_full` is determined by `word_cnt` greater than or equal to (*depth*+1)/2. The `almost_full` signal asserts when `word_cnt` is greater than or equal to '7'. This is the case since *af_from_top* is 1 when sets the `almost_full` threshold to '*depth* - `af_level`'. If *af_from_top* were set to 0, the setting of `af_level` of '2' would cause the `almost_full` flag to get asserted when `word_cnt` is greater than or equal to '2'.

Figure 1-8 Push to Full, Push and Pop While Full, Push Error

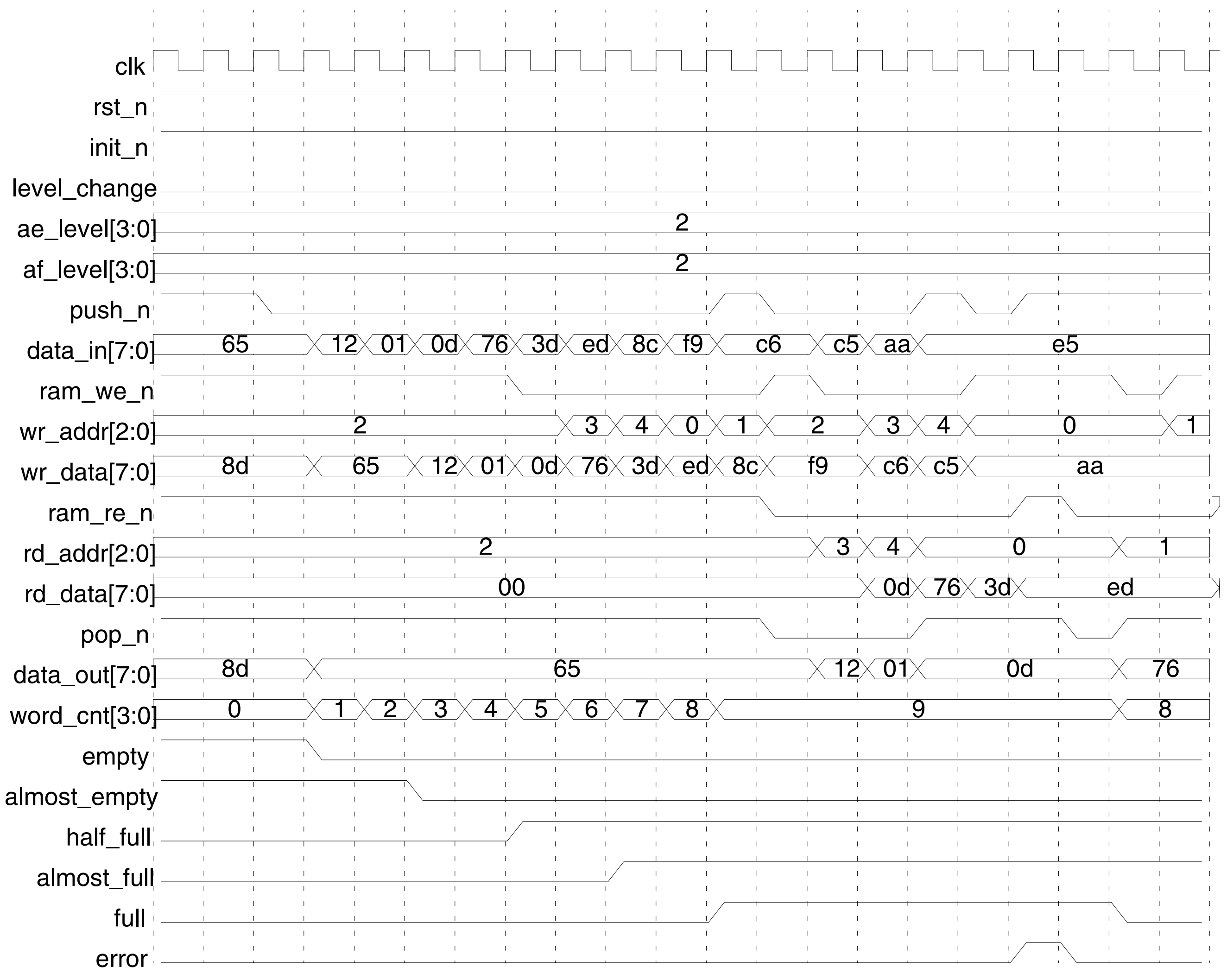


Figure 1-9 illustrates the condition in which the FIFO is empty and push and pop requests occur simultaneously.

Configuration: *width* is 8, *depth* is 8, *mem_mode* is 0, *arch_type* is 1, *af_from_top* is 1, *ram_re_ext* is 1, and *err_mode* is 0; pre-fetch cache depth of 1, no re-timing stages in RAM.

Again, in this configuration it is assumed that a synchronous RAM with no re-timing stages is connected to the DW_lp_fifoctl_1c_df and with the DW_lp_fifoctl_1c_df containing a 1-deep pre-fetch cache.

For this particular sequence, the pop on an empty FIFO causes the `error` flag to go active. However, the push request is legal which results in the `data_in` value of 0xf9 being written into the first (and only) stage of the pre-fetch cache. Therefore, on the next clock cycle following the sampled assertion of `pop_n`, the `data_out` contains the 0xf9. As a result, the `word_cnt` value is updated to 1. Note that the `error` flag once set to 1 stays asserted even though the 'illegal' pop request occurs for only one clock cycle. This is due to the *err_mode* setting of 0 which configures the `error` flag to maintain its asserted state until a system reset is performed.

Since the push request was allowed and `data_in` was written into cache, the second `pop_n` assertion (immediately following the illegal pop request) is legal and pops the word from the FIFO (or more specifically, from cache). This results in `word_cnt` going back to 0 and the FIFO `empty` status flag going active.

Figure 1-9 Push and Pop on Empty FIFO

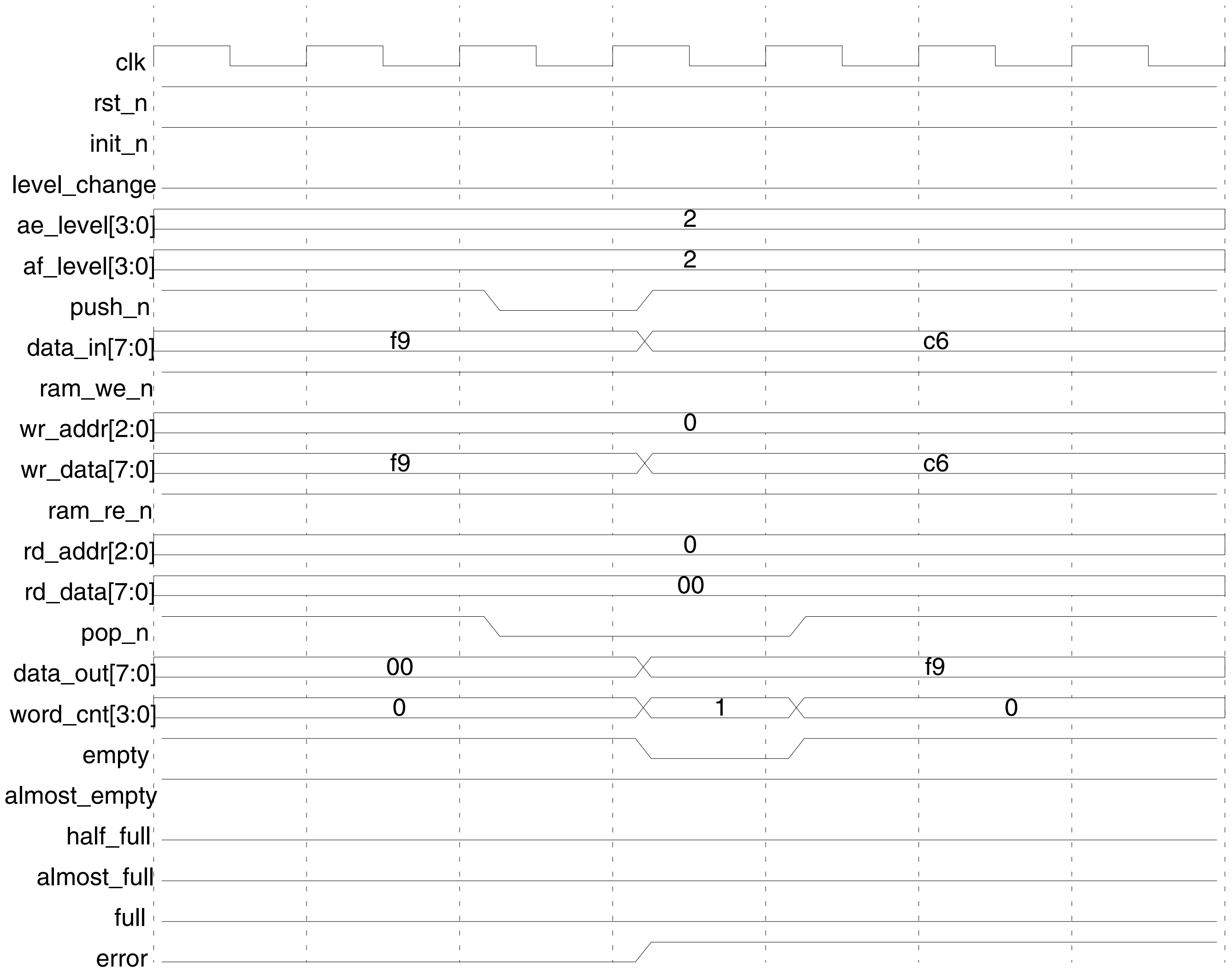


Figure 1-10 describes the affects changing '`ae_level`', '`af_level`', and '`level_change`' have on the '`almost_empty`' and '`almost_full`' flags.

Configuration: *width* is 8, *depth* is 8, *mem_mode* is 0, *arch_type* is 1, *af_from_top* is 1, *ram_re_ext* is 1, and *err_mode* is 0; pre-fetch cache depth of 1, no re-timing stages in RAM.

Again, in this configuration it is assumed that a synchronous RAM with no re-timing stages is connected to the DW_lp_fifoctl_1c_df with the DW_lp_fifoctl_1c_df containing a 1-deep pre-fetch cache.

Initially, the `ae_level` setting is '2' and `af_level` is '4'. From the FIFO empty state, three active cycles of `push_n` result in the FIFO to have three words written into it. This is reflected by `word_cnt` settling on '3'. Since `ae_level` is '2', the `almost_empty` flag stays asserted while `word_cnt` is '2' and less. Upon the `word_cnt` going to '3', `almost_empty` de-asserts.

After the three consecutive push requests, the FIFO remains idle until `ae_level` changes from '2' to '4'. The change in `ae_level` alone does not render a change in the `almost_empty` flag as evident in the `word_cnt` compared to the `ae_level` value not matching the state of the `almost_empty` flag in the cycles that follow. If the state of the `almost_empty` flag is critical between push and pop operations on a non-empty FIFO, the `level_change` input causes the `almost_empty` (and `almost_full`) flag to get updated on the next clock cycle. This is clearly seen in the waveform below once the `level_change` input is sampled to be 'high'. The `almost_empty` flag goes to 1 on the following clock cycle to reflect the state brought on by the newly changed value of '4' on the `ae_level` input.

Following the pulse of `level_change`, a single push request is issued. This adds another word written to the FIFO which increases the `word_cnt` to '4'. At this point, `half_full` goes to 1 since the FIFO depth is '8' and `almost_full` also goes to 1 since `word_cnt` is equal to or less than the `almost_full` threshold of '4' (*depth* - `af_level`).

Following this push request, the `af_level` changes from '4' to '2'. Again, with no push and pop activity the status flags will not update solely on changing `af_level` and, most notably, the `almost_full` will be out of sync in relation to `word_cnt` and the `af_level` values. That is, the `word_cnt` is '4' but the `almost_full` threshold value after `af_level` changed to '2' is '6' (*depth* - `af_level`). It is not until the next sampled assertion of `level_change` (described above) or a push or pop request that the `almost_full` reflects the change in `af_level`. In this example, `push_n` going to 0 causes `word_cnt` to update to '5' and the `almost_full` to 0 on the following cycle.

Note that the `almost_full` threshold is determined by '*depth* - `af_level`' because the parameter *af_from_top* is 1.

Figure 1-10 Behavior for almost_empty and almost_full

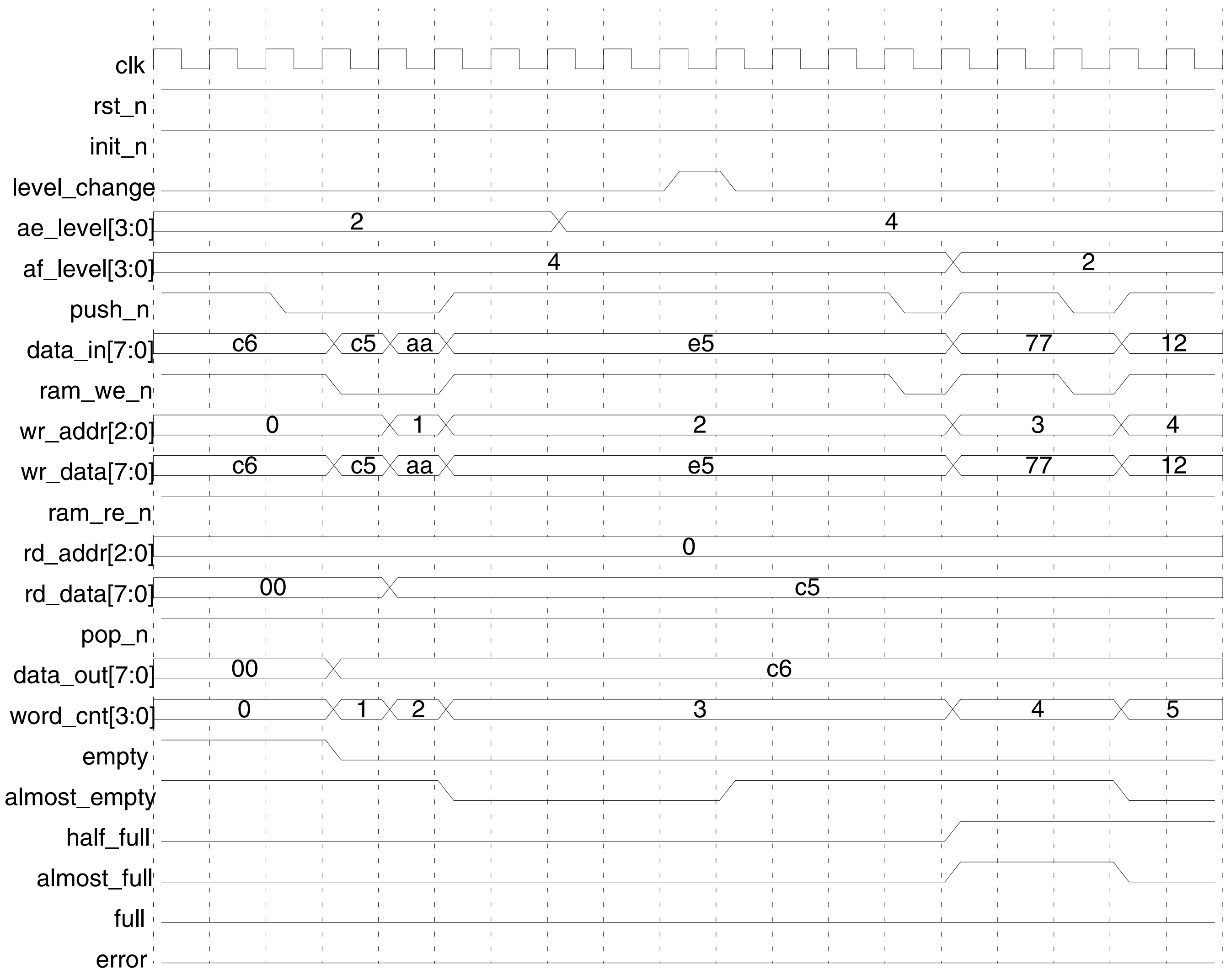


Figure 1-11 on page 29 shows the data path bypass characteristics of the pre-fetch.

Configuration: *width* is 8, *depth* is 8, *mem_mode* is 7, *arch_type* is 3, *af_from_top* is 0, *ram_re_ext* is 0, and *err_mode* is 0; pre-fetch cache depth of 3, RAM write, read address and data out re-timing stages.

This set of waveforms illustrates pushing and popping activity that solely utilizes the pre-fetch cache and not the RAM. Since the pre-fetch cache contains 3 locations for data (3 deep), the first 3 push requests from the FIFO empty state go directly to it. Notice that during these three pushing operations that `ram_we_n` does not get asserted (0) across the clock boundaries.

Once the `word_cnt` is '3', simultaneous push and pop requests are issued for 2 consecutive cycles. At this point, all the word transfers are going through the pre-fetch cache and no RAM accesses are initiated as evident by `ram_we_n` and `ram_re_n` remaining at 1.

The final three pop operations (`pop_n` at 0) are issued which empties the FIFO.

If the fourth push operation had occurred without a simultaneous pop request, the fourth push would have caused the RAM to get written with `data_in` contents. At that point, all the subsequent push operations would go through the RAM and not directly to the pre-fetch cache until the RAM eventually became empty again.

Figure 1-11 Pushing and Popping Through Full Pre-fetch Cache

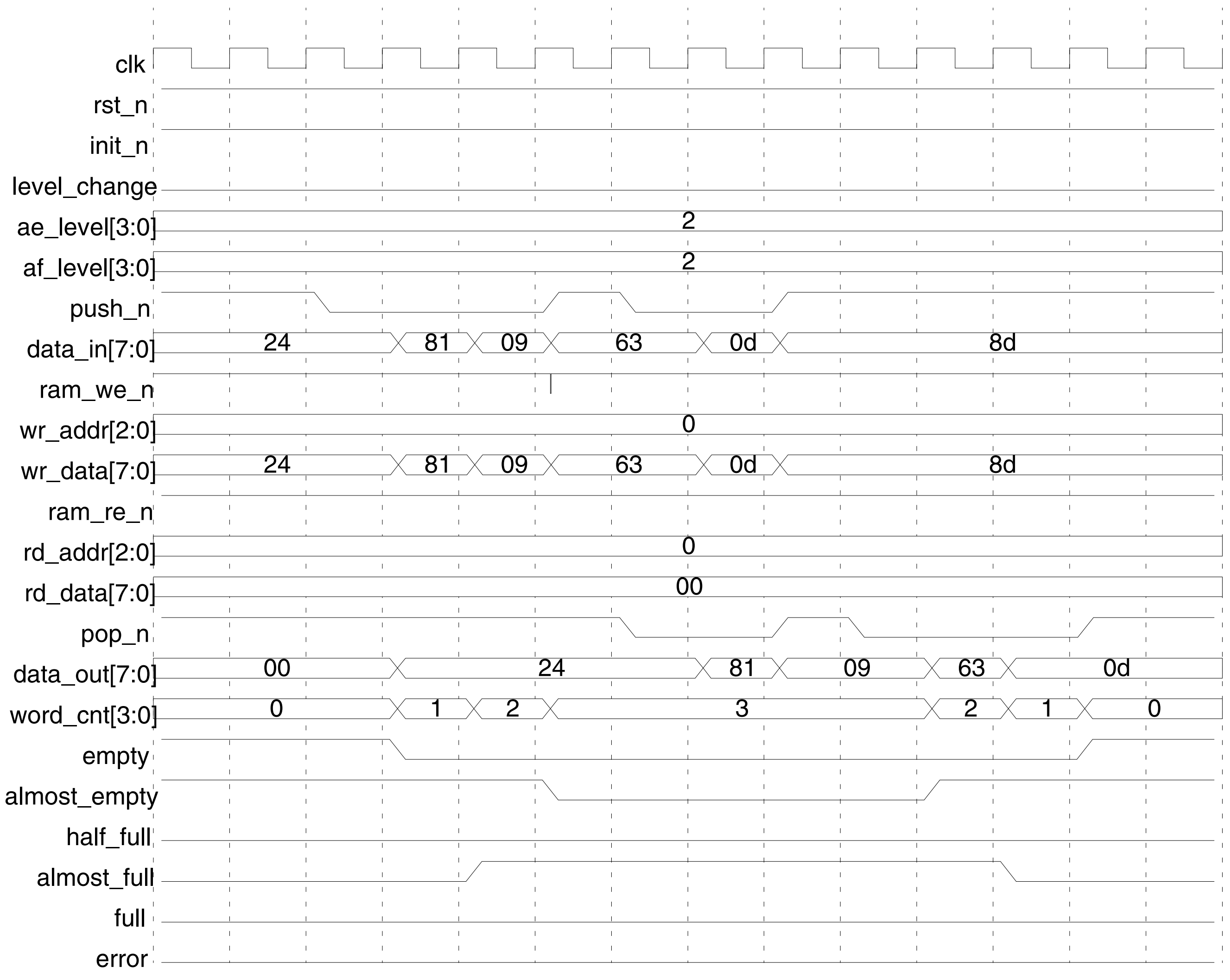


Figure 1-12 shows the affects of `init_n` for the configuration where no pre-fetch cache exists (*arch_type* is 0). When *arch_type* is 0, *mem_mode* must also be 0.

Configuration: *width* is 8, *depth* is 8, *mem_mode* is 0, *arch_type* is 0, *af_from_top* is 0, *ram_re_ext* is 1, and *err_mode* is 0; no pre-fetch cache, no re-timing stages in RAM.

The `init_n` is the synchronous reset for DW_lp_fifoctl_1c_df. Only when `init_n` is 0 and captured by the rising edge of `clk` will the DW_lp_fifoctl_1c_df sequential elements get initialized; this shown by two assertions of `init_n` in Figure 1-12. The occurrence of `init_n` going to 0 does not get sampled by the rising edge of `clk`. Therefore, the sequential elements in the DW_lp_fifoctl_1c_df are unaffected. However, the second pulse of `init_n` spans across the rising edge boundary of `clk`. This causes all the registers to be initialized to their default values. Most notably, `word_cnt` goes to 0, `half_full` and `almost_full` go to 0, and `empty` and `almost_empty` go to 1.

Figure 1-12 System Reset Using `init_n`

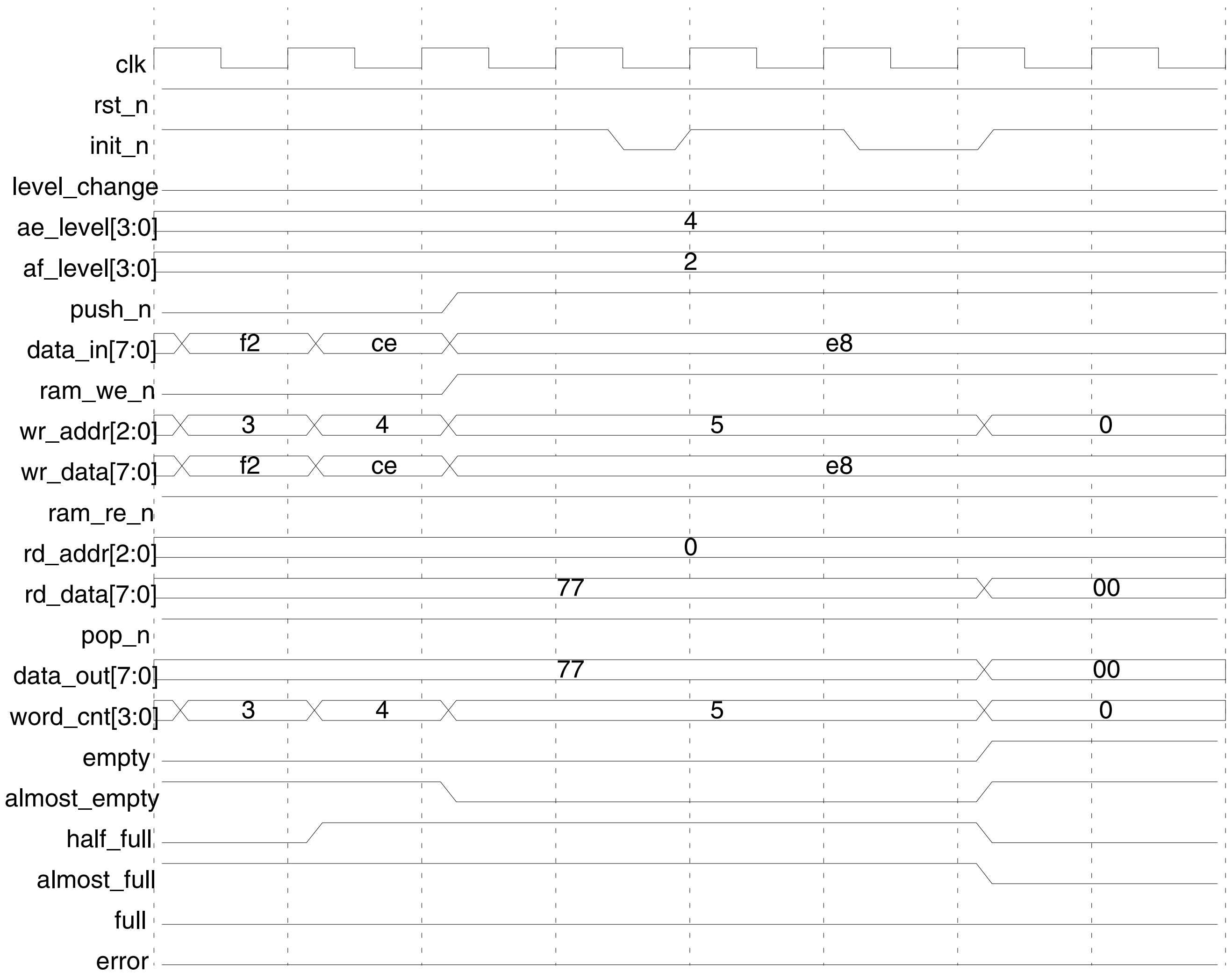
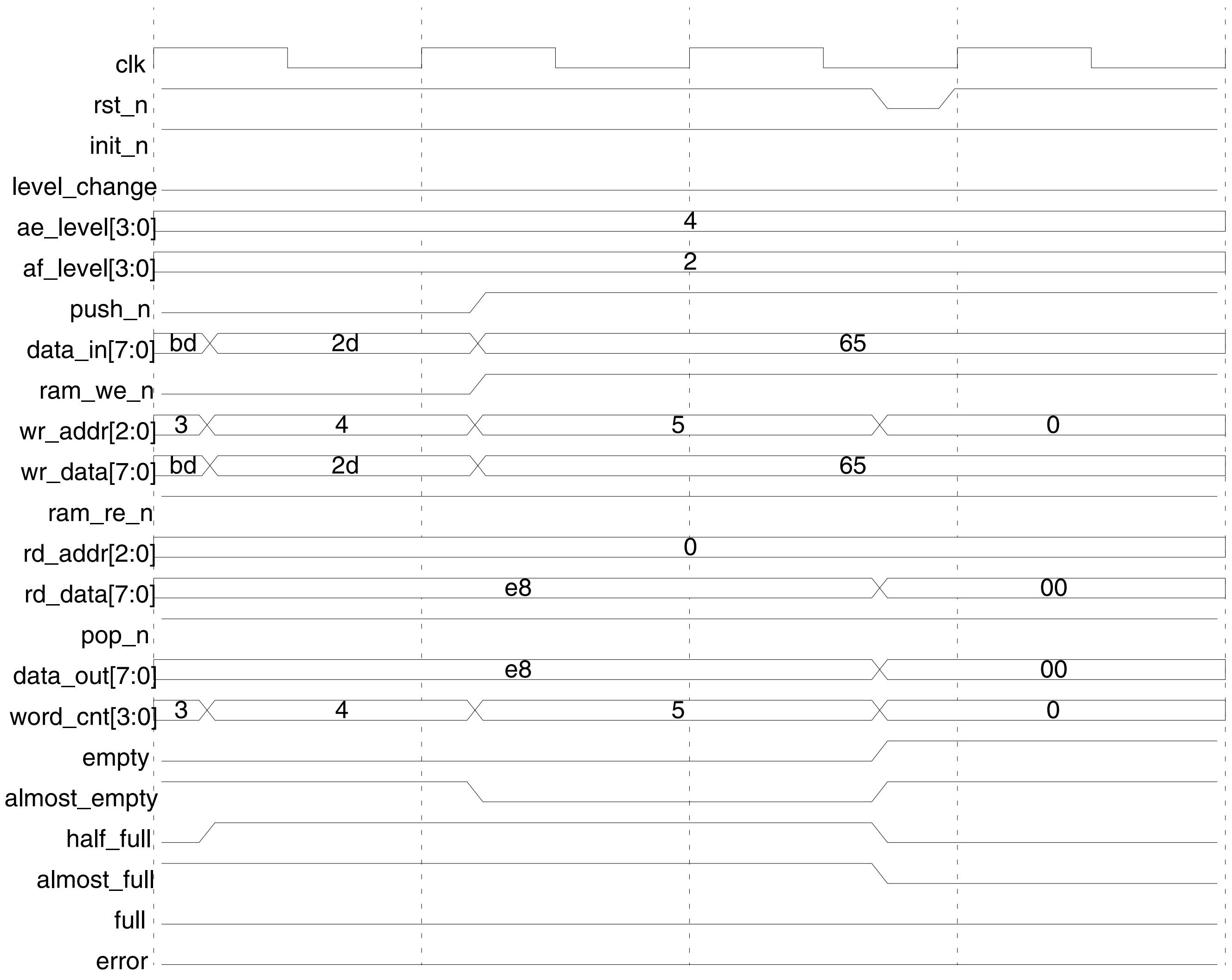


Figure 1-13 shows the affects of `rst_n` for the configuration where no pre-fetch cache exists (*arch_type* is 0). When *arch_type* is 0, *mem_mode* must also be 0.

Configuration: *width* is 8, *depth* is 8, *mem_mode* is 0, *arch_type* is 0, *af_from_top* is 0, *ram_re_ext* is 1, and *err_mode* is 0; no pre-fetch cache, no re-timing stages in RAM.

The `rst_n` is the asynchronous reset for DW_lp_fifoctl_1c_df. Whenever `rst_n` goes to 0 the DW_lp_fifoctl_1c_df sequential elements get initialized. From the waveforms below, as soon as the `rst_n` goes to 0 all the registers to be initialized to their default values. Most notably, `word_cnt` goes to 0, `half_full` and `almost_full` go to 0, and `empty` and `almost_empty` go to 1.

Figure 1-13 System Reset Using `rst_n`



Related Topics

- Memory – FIFO Overview
- DesignWare Building Blocks User Guide

HDL Usage Through Component Instantiation - VHDL

```
library IEEE,DWARE;
use IEEE.std_logic_1164.all;
use DWARE.DWpackages.all;
use DWARE.DW_Foundation_comp.all;

entity DW_lp_fifoctl_1c_df_inst is
      generic (
        inst_width : POSITIVE := 8;
        inst_depth : POSITIVE := 8;
        inst_mem_mode : NATURAL := 3;
        inst_arch_type : NATURAL := 1;
        inst_af_from_top : NATURAL := 1;
        inst_ram_re_ext : NATURAL := 0;
        inst_err_mode : NATURAL := 0
        );
      port (
        inst_clk : in std_logic;
        inst_rst_n : in std_logic;
        inst_init_n : in std_logic;
        inst_ae_level : in std_logic_vector(3 downto 0);
        inst_af_level : in std_logic_vector(3 downto 0);
        inst_level_change : in std_logic;
        inst_push_n : in std_logic;
        inst_data_in : in std_logic_vector(inst_width-1 downto 0);
        inst_pop_n : in std_logic;
        inst_rd_data : in std_logic_vector(inst_width-1 downto 0);
        ram_we_n_inst : out std_logic;
        wr_addr_inst : out std_logic_vector(2 downto 0);
        wr_data_inst : out std_logic_vector(inst_width-1 downto 0);
        ram_re_n_inst : out std_logic;
        rd_addr_inst : out std_logic_vector(2 downto 0);
        data_out_inst : out std_logic_vector(inst_width-1 downto 0);
        word_cnt_inst : out std_logic_vector(3 downto 0);
        empty_inst : out std_logic;
        almost_empty_inst : out std_logic;
        half_full_inst : out std_logic;
        almost_full_inst : out std_logic;
        full_inst : out std_logic;
        error_inst : out std_logic
        );
    end DW_lp_fifoctl_1c_df_inst;

architecture inst of DW_lp_fifoctl_1c_df_inst is

begin
    -- Instance of DW_lp_fifoctl_1c_df
    U1 : DW_lp_fifoctl_1c_df
    generic map ( width => inst_width,
                      depth => inst_depth,
                      mem_mode => inst_mem_mode,
                      arch_type => inst_arch_type,
                      af_from_top => inst_af_from_top,
                      ram_re_ext => inst_ram_re_ext,
                      err_mode => inst_err_mode
                    )
    port map ( clk => inst_clk,
                   rst_n => inst_rst_n,
                   init_n => inst_init_n,
                   ae_level => inst_ae_level,
                   af_level => inst_af_level,
                   level_change => inst_level_change,
                   push_n => inst_push_n,
                   data_in => inst_data_in,
                   pop_n => inst_pop_n,
                   rd_data => inst_rd_data,
                   ram_we_n => ram_we_n_inst,
                   wr_addr => wr_addr_inst,
                   wr_data => wr_data_inst,
                   ram_re_n => ram_re_n_inst,
                   rd_addr => rd_addr_inst,
                   data_out => data_out_inst,
                   word_cnt => word_cnt_inst,
                   empty => empty_inst,
                   almost_empty => almost_empty_inst,
                   half_full => half_full_inst,
                   almost_full => almost_full_inst,
                   full => full_inst,
                   error => error_inst
                 );


end inst;

-- Configuration for use with a VHDL simulator
-- pragma translate_off
library DW03;
configuration DW_lp_fifoctl_1c_df_inst_cfg_inst of DW_lp_fifoctl_1c_df_inst is
  for inst
  end for; -- inst
end DW_lp_fifoctl_1c_df_inst_cfg_inst;
-- pragma translate_on
```

HDL Usage Through Component Instantiation - Verilog

```
module DW_lp_fifoctl_1c_df_inst ( inst_clk, inst_rst_n, inst_init_n,
           inst_ae_level, inst_af_level, inst_level_change, inst_push_n,
           inst_data_in, inst_pop_n, inst_rd_data, ram_we_n_inst,
           wr_addr_inst, wr_data_inst, ram_re_n_inst, rd_addr_inst,
           data_out_inst, word_cnt_inst, empty_inst, almost_empty_inst,
            half_full_inst, almost_full_inst, full_inst, error_inst
                             );

parameter width       = 8;
parameter depth       = 8;
parameter mem_mode    = 3;
parameter arch_type   = 1;
parameter af_from_top = 1;
parameter ram_re_ext  = 0;
parameter err_mode    = 0;

`define cnt_width   4  // log2(depth+1)
`define addr_width  3  // log2(ram_depth), ram_depth = 5


input                            inst_clk;
input                            inst_rst_n;
input                            inst_init_n;
input  [`cnt_width-1:0]          inst_ae_level;
input  [`cnt_width-1:0]          inst_af_level;
input                            inst_level_change;
input                            inst_push_n;
input  [width-1:0]               inst_data_in;
input                            inst_pop_n;
input  [width-1:0]               inst_rd_data;

output                           ram_we_n_inst;
output [`addr_width-1:0]         wr_addr_inst;
output [width-1:0]               wr_data_inst;
output                           ram_re_n_inst;
output [`addr_width-1:0]         rd_addr_inst;
output [width-1:0]               data_out_inst;
output [`cnt_width-1:0]          word_cnt_inst;
output                           empty_inst;
output                           almost_empty_inst;
output                           half_full_inst;
output                           almost_full_inst;
output                           full_inst;
output                           error_inst;
```

```
DW_lp_fifoctl_1c_df #(width, depth, mem_mode, arch_type, af_from_top, ram_re_ext,
err_mode) U1 (
            .clk(inst_clk),
            .rst_n(inst_rst_n),
            .init_n(inst_init_n),
            .ae_level(inst_ae_level),
            .af_level(inst_af_level),
            .level_change(inst_level_change),
            .push_n(inst_push_n),
            .data_in(inst_data_in),
            .pop_n(inst_pop_n),
            .rd_data(inst_rd_data),
            .ram_we_n(ram_we_n_inst),
            .wr_addr(wr_addr_inst),
            .wr_data(wr_data_inst),
            .ram_re_n(ram_re_n_inst),
            .rd_addr(rd_addr_inst),
            .data_out(data_out_inst),
            .word_cnt(word_cnt_inst),
            .empty(empty_inst),
            .almost_empty(almost_empty_inst),
            .half_full(half_full_inst),
            .almost_full(almost_full_inst),
            .full(full_inst),
            .error(error_inst)
            );

endmodule
```

Revision History

For notes about this release, see the *DesignWare Building Block IP Release Notes*.

For lists of both known and fixed issues for this component, refer to the STAR report.

For a version of this datasheet with visible change bars, click here.

Date	Release	Updates
July 2020	DWBB_201912.5	■ Adjusted content and title of "Suppressing Warning Messages During Verilog Simulation" on page 22 and added the DW_SUPPRESS_WARN macro
October 2019	DWBB_201903.5	■ Added the "Disabling Clock Monitor Messages" section
March 2019	DWBB_201903.0	■ Clarified some information about minPower in Table 1-5 on page 5 ■ Added this Revision History table and the document links on this page